Railway System User Manual

Revision 1.1

Tobias Simon

November 19, 2009

Contents

1 Overview **1**

2 Hardware and Base Software **1**

3 Steering Implementation **3**

3.1 The AbstractSteering class 3

3.2 Avoiding realtime programming pitfalls 5

3.3 System requirements for x86 realtime 6

4 RSAPI reference **7**

4.1 Power Interface 8

4.2 Reed Contact Interface 9

4.3 Turnout Interface 10

4.4 Locomotive Interface 11

4.5 Communication Interface 12

1 Overview

This manual gives a brief introduction on how to implement a model railroad system steering with the supplied base software. First, the hardware is described in Section 2. After that, a short guide on programming the steering with the "`AbstractSteering`" class is given in Section 3. This section also issues some do's and dont's for implementing realtime systems with the RTSJ ("Real-Time Specification for Java"). Finally, the programming interface to the railway system is defined in Section 4.

2 Hardware and Base Software

The railway system is based on a ground plate on which all railway system components are mounted and wired. The locomotives drive on the rails which are interconnected by a crossing in the center and four turnouts, as shown in Figure 1. To provide information about the locomotives' positions, 16 reed

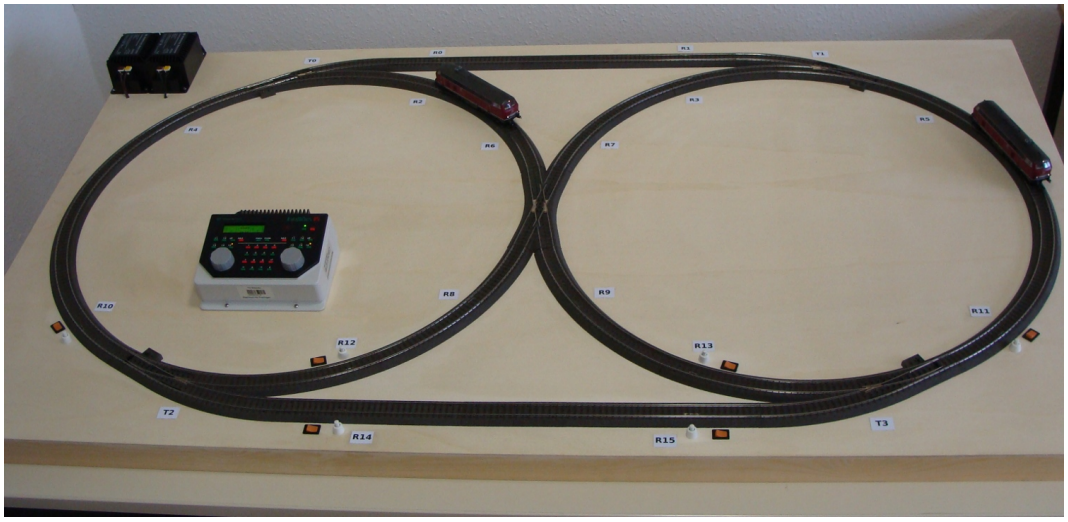

Figure 1: Image of the railway system

contacts are installed on the track bed. To simulate temporary erroneous reed contact activations, pushbuttons are installed next to the six foremost reed contacts. Additionally, six rocker switches are installed. They are used to

disable the corresponding reed contacts in order to simulate a permanent reed contact failure. The whole system is controlled by a serial communication protocol (IEA232) driven by an RSAPI backend. The programming environment resides on a standard PC and includes the following base software:

- Realtime Linux based on Ubuntu
- Sun JRTS including JVM/JDK
- Eclipse IDE
- Eclipse workspace

The Eclipse workspace contains all software components needed to implement a steering. It includes the RSAPI, the `AbstractSteering` class and a simple example steering called "`RTT_Steering`". The complete set-up including railway system and PC is shown in Figure 2.

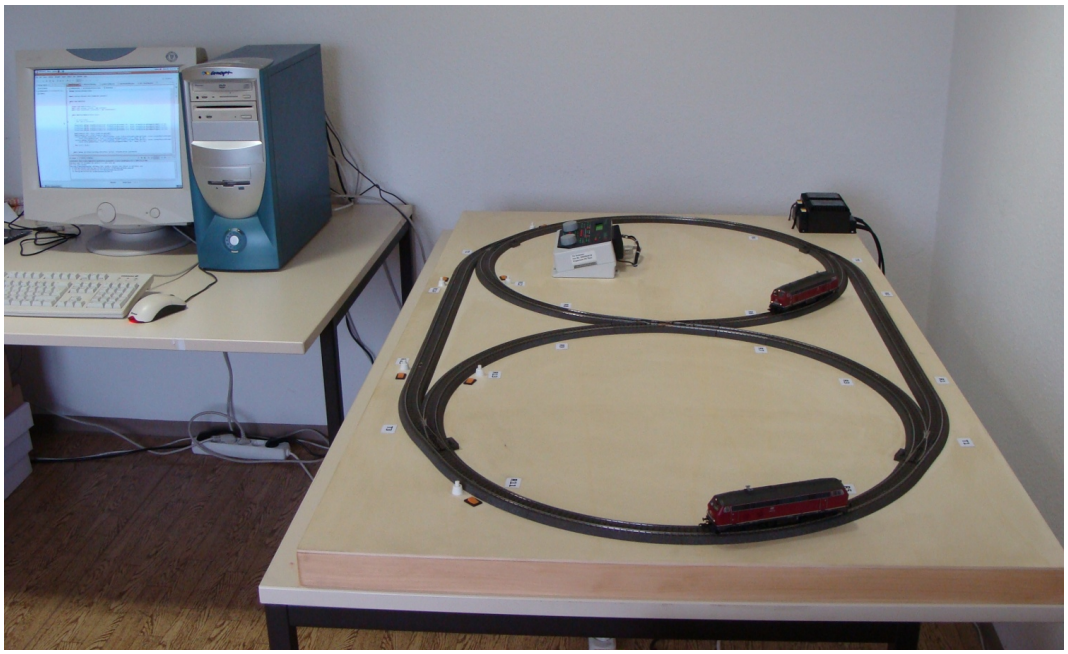

Figure 2: Railway system and PC

3 Steering Implementation

The realtime steering uses the AbstractSteering class to encapsulate all workflow functionality. In order to write a custom steering, the user needs to derive his steering class from "`AbstractSteering`". User code should reside in the "`steering`" package. To form a cleaner namespace an additional sub-package, which holds the user implementation, may be created.

3.1 The AbstractSteering class

The `AbstractSteering` class provides some features which are useful to the programmer. First of all, it provides an instance of the RSAPI (field: "`api`") used to interact with the railway system (see Section 4 for details).

When the system starts up (time from starting the program till the control flow reaches user code), the `AbstractSteering` initializes all locomotives and turnouts to their default states which are defined by the RSAPI (see Section 4.3 and 4.4). It provides exactly one entry point for the whole control flow of the derived steering class. This entry point is the abstract method "`drive()`", which is called after the system initialization is complete. For information about its class context, take a look at Figure 3. This figure also states that `AbstractSteering` is essentially a `NoHeapRealTimeThread`.

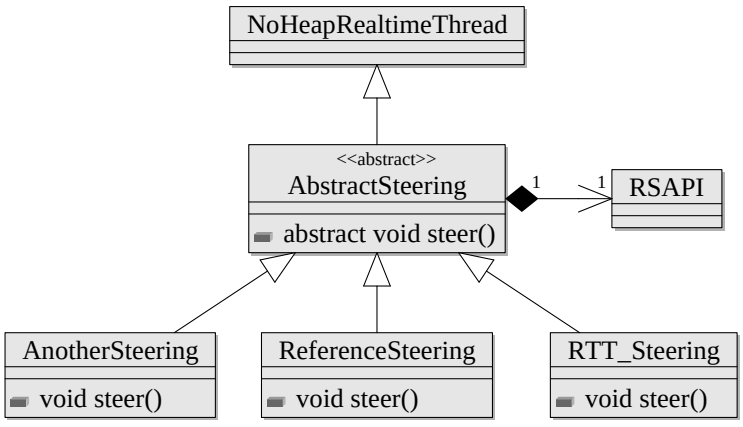

Figure 3: `AbstractSteering` class diagram

To make the steering secure, it implements the following behaviour: Any

exception that leaves `drive()` up the call stack will put the railway system in a safe state (power-off). Relevant error and/or warning messages will be written to the standard error output. A minimal example of how to structure the steering is shown in Listing 1.

```
1  public class StubSteeing extends AbstractSteering
2  {
3     /* Place your private fields here. */
4
5     public StubSteering(RSAPI api) throws Exception
6     {
7        super(api);
8        /* Place memory allocation (object creation) here.
9           Note: do not use the api at this stage or the
10          AbstractSteering initialization sequence may fail! */
11    }
12
13    @Override
14    public void steer() throws Exception
15    {
16       /* Perform initial api operations here,
17          e.g. start locomotives. */
18
19       /* steering loop: */
20       while (true)
21       {
22          /* Read reed contact states / perform steering step.
23             Note: Do not allocate any memory here,
24                   possible memory leak! */
25       }
26    }
27 }
```

Listing 1: Stub steering implementation

In addition, a real example is provided in the `steering` package. The Example is called "`RTT_Steering`" and just measures the round trip time of locomotive 1, which is instructed to drive with a certain speed.

3.2 Avoiding realtime programming pitfalls

Since a hard realtime system is being developed, special problems must be taken into concideration:

- The `Main` class, which creates the steering instance, boosts the priority of all JVM tasks above any process in the system, except the interrupt handlers. A programming mistake may lead to a deadlocking program which results in a system which no more responds to keyboard or mouse stimuli. In this case, power-cycling is needed to get the PC operational again. To avoid this, a defensive programming attitude is required: The main steering loop must give other processes some CPU ticks either through `NoHeapRealtimeThread.sleep()` or through `NoHeapRealtimeThread.waitForNextPeriod()` to avoid the continuous preemtion of these processes.

- Do not allocate memory or call standard-library functions which may allocate memory in the steering loop. This will cause a memory leak, since a `NoHeapRealtimeThread` is always located in `ImmortalMemory`. This memory area allows allocating memory which is never freed by the garbage collector. The consequence is that all steering-relevant objects and data structures need to be allocated before the steering loop is entered. See also the comments in Listing 1.

- Do not write directly to a file – the realtime behaviour will depend on the harddisk access speed and the I/O scheduler. On the other side, pagefaults are occuring when files are accessed under Linux, which may also influence the realtime behaviour. If writing to a file is needed, use a seperate non-realtime thread for this purpose and let the realtime thread exchange data with it using a `WaitFreeWriteQueue`.

An example showing the combination of a `NoHeapRealTimeThread` and a `WaitFreeWriteQueue` to communicate with a standard Thread is located in the "`rt-latency`" project in the workspace. It measures the latency and jitter of the given NoHeapRealtimeThread using a periodic sleep of 1ms. Please take also a look at the file "`README.TXT`" located in the project. A negative example is given by the "`non-rt-latency`" project. It tries to make as much mistakes as possible to show you what not to do in a realtime software project.

3.3 System requirements for x86 realtime

The following points are necessary in order to enable realtime programming on an x86 PC architecture:

- use an RT-enabled Linux-Kernel
- use the modified "/etc/security/limits.conf":

```
@realtime soft cpu unlimited
@realtime - rtprio 100
@realtime - nice 40
@realtime - memlock unlimited
```

- enable the "performance" CPU frequency scaler: "cpufreq-set -g performance"
- disable the SMI if present ("smictrl -s 0")
- use PS/2 mouse and keyboard
- do not plug-in an USB-stick before powering on the machine

4 RSAPI reference

The "RSAPI" is an interface which is used to interact with each controllable or accessible element of the railway system. The acronym stands for "Railway System Application Programming Interface".

The interface introduces an abstraction which helps to separate logical elements like *locomotives*, *turnouts* and r*eed contacts* from physical decoder adressing conciderations. To identify each individual logical element, the enumerator classes "`Locomotive`", "`Turnout`" and "`ReedContact`" reside in the interface package (`rsapi`). The enumerator classes contain the following enumerator identifiers:

- `Locomotive` → `Locomotive.L0` and `Locomotive.L1`
- `Turnout` → `Turnout.T0` ... `Turnout.T3`
- `ReedContact` → `ReedContact.R0` ... `ReedContact.R15`

The turnout and reed contact identifiers are mapped to physical railway system elements as stated in Figure 4. Locomotive identifiers are mapped to both physical locomotives, according to the identifiers on their sides. The in-

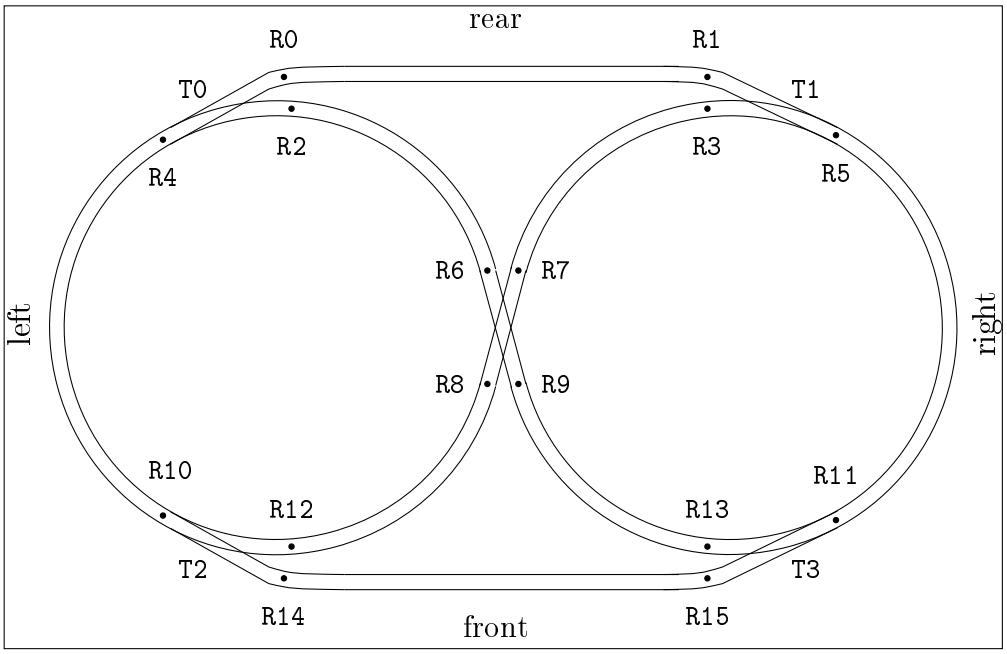

Figure 4: Identifier mapping on the railway system

terface is used to control different entities of the railway system. Based on this separation, the following five independend interface parts are established:

- Power Interface (section 4.1)
- Reed Contact Interface (section 4.2)
- Turnout Interface (section 4.3)
- Interface Locomotive (section 4.4)
- Communication Interface (section 4.5)

All methods in the RSAPI are blocking. That means, they either succeed or an exception is thrown. The maximum blocking time, as stated by the IntelliBox manual, is 50ms. If asynchronous behavior is required, additional threads may be implemented. For redundancy, it may be useful to create one thead for the steering and one for the communication with the other steering PC.

4.1 Power Interface

The Power Interface provides two methods for turning all power-consuming railway system components on and off. Please note that the locomotives and turnouts can't be steered while the power is off.

`void powerOn()` turns the power on.

`void powerOff()` turns the power off.

Both methods throw an exception in case of a backend or communication error.

4.2 Reed Contact Interface

This part of the interface is used to retrieve the reed contact states. It consists of the main methods `readReedContactStates` and `getReedContactState` and some additional utility methods. This interface is implemented stateful in order to improve throughput and to simplify the resulting programs.

`void readReedContactStates()` reads all 16 reed contact states from the hardware backend into an internal buffer. This method throws an exception in case of a backend or communication error.

`boolean getReedContactState(ReedContact contact)` returns the currently buffered state of the given reed contact. If `true` is returned, then the contact was closed (active) when `readReedContactStates` was called the last time. Otherwise, `false` is returned. If `readReedContactStates` has never been called before, the default reed contact state (`false`) is returned.

`double getReedContactLastChange(ReedContact contact)` returns the time (in milliseconds) since the buffered state of the given reed contact state has changed.

`boolean reedContactChanged(ReedContact contact)` returns `true` if the buffered reed contact state has changed between two subsequent calls of `readReedContactStates`.

`boolean reedContactClosed(ReedContact contact)` returns `true` if the buffered reed contact state has changed from "open" to "closed" between two subsequent calls of `readReedContactStates`.

`boolean reedContactOpened(ReedContact contact)` returns `true` if the buffered reed contact state has changed from "closed" to "open" between two subsequent calls of `readReedContactStates`.

4.3 Turnout Interface

The turnout interface is used to switch the individual turnouts. The state of each turnout is buffered, so inverting the turnout direction is possible without the otherwise needed extra application-level turnout state management. On startup, the states are initialized to `TurnoutDir.DIR_STRAIGHTON`. The descriptions of the interface methodes follow:

`void writeTurnoutState(Turnout turnout)` writes the buffered turnout state to the hardware. This method throws an exception in case of a backend or communication error.

`void setTurnoutState(Turnout turnout, TurnoutState state)` sets the buffered turnout state to a new value.

`TurnoutState getTurnoutState(Turnout turnout)` returns the buffered tunout state.

`public void invertTurnoutState(Turnout turnout)` inverts the currently buffered turnout state.

The interface does not perform any turnout switching on startup. The initialization is done by the `AbstractSteering` class. The turnouts are put in a known state as shown in the following listing 2.

```
1 RelativeTime turnoutSleepInterval = new RelativeTime(250,0);
2
3 for (Turnout turnout: Turnout.values())
4 {
5    /* Write default state */
6    api.writeTurnoutState(turnout);
7    /* Wait 250ms to let the turnout settle. */
8    NoHeapRealtimeThread.sleep(turnoutSleepInterval);
9 }
```

Listing 2: Turnout initialization sequence

4.4 Locomotive Interface

This part of the RSAPI interface provides support for controlling the both locomotives. This interface is stateful – the state is written to the locomotive decoder via `writeLocState`.

`void writeLocState(Locomotive locomotive, boolean writeFunctions)` writes the buffered locomotive state to the hardware. If `writeFunctions` is `true`, the buffered states of `LocFunction.F1` ... `LocFunction.F3` will also be written to the locomotive controller[1]. This method throws an exception in case of a backend or communication error.

[1]Since the current locomotives do not implement these functions, this flag has no influence.

`void setLocSpeed(Locomotive loc, LocSpeed speed)` sets the buffered speed for the locomotive.

`LocSpeed getLocSpeed(Locomotive loc)` returns the last speed which was set by `setLocomotiveSpeed` or `LocSpeed.SPEED_0` if `setLocomotiveSpeed` has never been called before.

`void setLocDirection(Locomotive loc, boolean forward)` sets the buffered locomotive direction. Note: The locomotives should only be driven in the backward direction, because the reed contact magnet is placed below the rear axis.

`boolean getLocDirection(Locomotive loc)` returns the buffered locomotive direction.

`void setLocFunction(Locomotive loc, LocFunction f, boolean enabled)` sets the buffered locomotive function state to the value of `enabled`.

`boolean getLocFunction(Locomotive loc, LocFunction function)` returns the buffered locomotive function state.

As in the turnout interface (see 4.3), the locomotive interface itself does not perform any locomotive initialization on startup. Putting the locomotives into a known state before executing the user code is also done by the `AbstractSteering` class (see Listing 3). The locomotive state after startup is: speed 0, direction forward, functions disabled.

```
1 for (Locomotive locomotive: Locomotive.values())
2 {
3    /* Write default locomotive state to the decoder. */
4    api.writeLocState(locomotive, true);
5 }
```

Listing 3: Locomotive initialization sequence

4.5 Communication Interface

The last part of the RSAPI interface allows to communicate with another steering PC over a serial communication channel. This port is directly beneath the serial port, which is interconnected with the railway system. This interface is only needed for implementing hot-standby redundancy in the steering program. It may be used concurrent to the other interface parts by another thread because it accesses a dedicated communication interface.

`void writeCharToPC(char c)` sends a character to the other steering PC.

`char readCharFromPC()` receives a character from the other steering PC.

Both methods throw an exception in the case of a communication error.